

Exposition: Synthesis via Functional Interpretation

Daniel Weller

March 2, 2014

The aim of this short paper is to give a practical introduction to functional interpretation of proofs in arithmetic for computer scientists interested in synthesis. Towards this, we will define our own notion of functional interpretation which differs (sometimes only inessentially) from those used in the literature, but has the advantage (in the opinion of the author) of being very natural. Note that we only show functional interpretation by definition and example — it is quite possible that proving the correctness of this formalism is cumbersome (or even impossible). Still, as can be witnessed below, our formalism allows extraction of a correct program from a non-trivial proof in a systematic way, hopefully elucidating a few central ideas common to all notions of functional interpretation.

With this disclaimer in mind, we can start setting up our machinery for functional interpretation. Towards extracting a program from a proof, we have to fix (1) a programming language, (2) a proof system, and (3) a translation from (2) to (1).

The programming language. Choosing a suitable programming language for functional extraction involves some design decisions. Usually, one takes a functional programming language, since this often induces a simple and natural translation from proofs to programs. Some other decisions are more inessential, e.g. how to represent boolean values in the programming language (e.g. by constants \perp , \top , or by numerals $0, 1$).

For simplicity, we choose the untyped λ -calculus¹ extended with some constants for arithmetic, pairs, and program control. More precisely, we assume existence of a countable set of variables V , and a fixed set of constant symbols $\{0, 1, +, \div, \mathbf{pair}, \mathbf{left}, \mathbf{right}, \mathbf{isZero}, \mathbf{IfThenElse}, \mathbf{R}\}$. Variables and constants are λ -terms, and if s, t are λ -terms and x a variable then st and $\lambda x.t$ are λ -terms. We use infix notation for $+$, \div and write $\mathbf{IfThenElse}_{t_1 t_2 t_3}$ as $\mathbf{If } t_1 \mathbf{ Then } t_2 \mathbf{ Else } t_3$ and $\mathbf{pair}_{t_1 t_2}$ as (t_1, t_2) . The intended semantics of the symbols are clear except maybe for \mathbf{R} , which will be the recursion operator. Variable-free terms consisting only of $0, 1, +$ may denote numbers (i.e. $1 + (1 + 1)$ and $(1 + 1) + 1$ both denote $3 \in \mathbb{N}$, while $+++$ does not denote a number). Such terms are called *arithmetical*, and we will often not distinguish between a number $\alpha \in \mathbb{N}$ and the arithmetical terms that denote it. In particular, if $\alpha \in \mathbb{N}$ and t is a λ -term, then by $t\alpha$ we denote the λ -term which applies t to the numeral representing α .

¹All the λ -terms obtained by functional extraction are actually typable. We chose to use the untyped λ -calculus since in the context of this exposition, we view types as a distraction. Most works on functional extraction *do* work with a typed λ -calculus, since it is useful in e.g. proving correctness.

The formal semantics of our programming language are given by the reduction rules:

$$\begin{aligned}
& (\lambda x.t)s \rightarrow t[x := s], \\
& \mathbf{left}(s, t) \rightarrow s, \\
& \mathbf{right}(s, t) \rightarrow t, \\
& \mathbf{isZero}(0) \rightarrow 0, \\
& \mathbf{isZero}(t + 1) \rightarrow 1, \\
& \mathbf{If } 0 \mathbf{ Then } t \mathbf{ Else } s \rightarrow t, \\
& \mathbf{If } 1 \mathbf{ Then } t \mathbf{ Else } s \rightarrow s, \\
& \mathbf{Rbs}0 \rightarrow b, \\
& \mathbf{Rbs}(t + 1) \rightarrow st(\mathbf{Rbst}), \\
& t \dot{-} s \rightarrow u \quad \text{if } t, s \text{ arithmetical denoting } a, b, \text{ and } a > b, \text{ and } u \text{ denotes } a - b, \\
& t \dot{-} s \rightarrow 0 \quad \text{if } t, s \text{ arithmetical denoting } a, b, \text{ and } a \leq b,
\end{aligned}$$

where $[x := s]$ denotes capture-avoiding substitution. Hence we have β -reduction and the usual defining reductions for our constant symbols, where we have chosen to represent “true” by 0 and “false” by 1. Note that, in the clauses for \mathbf{R} , the term b corresponds to the base case of a recursive definition, the term s corresponds to the step case, and s will usually be of the form $\lambda x \lambda y. t(x, y)$, where the variable x corresponds to the recursion counter and the variable y to the result of the recursive call. $\dot{-}$ denotes the usual „cutoff subtraction” on the natural numbers.

It is fair to call this system a programming language: the set of terms is recursive and the relation \rightarrow has low computational complexity. It is easy to see that an interpreter for this language (i.e. an implementation of the transitive, reflexive, compatible closure of the \rightarrow relation) can be written in any Turing-complete programming language.

Proof system. We use natural deduction for intuitionistic logic with equality and induction (over the language $\{0, 1, +, =, \geq\}$). For the sake of conciseness, we only present the subset of rules that we will use in the example presented later in this paper.

$$\begin{array}{c}
[A] \quad [B] \\
\frac{A \quad B}{A \wedge B} \wedge_i \quad \frac{A \vee B \quad \begin{array}{c} \vdots \\ C \end{array} \quad \begin{array}{c} \vdots \\ C \end{array}}{C} \vee_e \quad \frac{A}{A \vee B} \vee_i \quad \frac{B}{A \vee B} \vee_i \quad \frac{A \quad A \rightarrow B}{B} \rightarrow_e \\
[B] \\
\frac{A}{\forall x.A} \forall_i \quad \frac{\forall x.A}{A[x := t]} \forall_e \quad \frac{\exists x.B \quad \begin{array}{c} \vdots \\ A \end{array}}{A} \exists_e \quad \frac{A[x := t]}{\exists x.A} \exists_i \\
[A(x)] \\
\frac{A(0) \quad \begin{array}{c} \vdots \\ A(x+1) \end{array}}{\forall x.A} \text{IND} \quad \frac{t = s \quad A(s)}{A(t)} =
\end{array}$$

where, as usual, $[A]$ denotes discharging an assumption A and the \exists_e, \forall_i rules have an eigenvariable condition. At the leaves of trees constructed by these rules, we allow only discharged assumptions and (non-discharged) *axioms*, which we take to be $t = t$, $t \geq t$, $t = 0 \vee \exists y. t = y + 1$, $t \geq s \rightarrow t + 1 \geq s + 1$, and $t \geq 0$ for all terms s, t .

Note that while our proof system is not directly suitable for automated proof search (in contrast to e.g. resolution proof systems), most other proof systems can be polynomially translated into our system.

Program extraction. We will now define a map \mathcal{E} from proofs to λ -terms with the intention that for a proof π of $\forall x \exists y. F(x, y)$ we will have² $\mathbb{N} \models F(n, \mathcal{E}(\pi)(n))$ for all $n \in \mathbb{N}$, where \models is the usual semantic

²This is not precisely true: actually, $\mathcal{E}(\pi)(n)$ will be a pair s.t. $\mathbb{N} \models F(n, \mathbf{left}(\mathcal{E}(\pi)(n)))$.

consequence operator (which in particular interprets our λ -terms as functions in the natural way). In other words, $\mathcal{E}(\pi)$, when viewed as function $\mathbb{N} \rightarrow \mathbb{N}$, fulfills the specification F .

The idea in defining \mathcal{E} will be to construct the “computational content” of a proof from the computational content of its premises. The most basic idea is that when we have a proof of B from an assumption $[A]$, then the computational content of B will depend upon that of A . In our setting, this means that the assumption $[A]$ induces a variable x_A in the computational content of B , and this variable will at some point be substituted by some other computational content as determined by the proof. In general, the type of the computational content will be closely related to the formula that is derived; for example a proof of $\exists x.F$ will have as computational content a pair (t, c) where t is the witness of $\exists x$, and c is the computational content of the proof of $F[x := t]$. Note that, unsurprisingly, the computational interpretations of \exists and \vee are closely related, as are the interpretations of \forall and \wedge . Roughly speaking, the propositional structure of the proof determines the structure of *functionals and control* in the program, while the quantifiers determine the structure of *data* in the program.

It is immediately clear that some proofs do not contain computational content (e.g. a proof of $A \rightarrow A$)³, therefore it will be useful to fix a variable ε which we will use to denote “no computational content”.⁴

The map \mathcal{E} is defined by structural induction on natural deduction proofs. For discharged assumptions, we set $\mathcal{E}([A]) := x_A$ (i.e. all discharged assumptions A are assigned the same variable x_A). For the axioms, we mostly assign no computational content by setting $\mathcal{E}(t = t) := \varepsilon$, $\mathcal{E}(t \geq t) := \varepsilon$ and $\mathcal{E}(t \geq s \rightarrow t + 1 \geq s + 1) := \lambda x.\varepsilon$, except that we set

$$\mathcal{E}(t = 0 \vee \exists y.t = y + 1) := (\text{isZero}(t), \text{If isZero}(t) \text{ Then } \varepsilon \text{ Else } (t \div 1, \varepsilon)).^5$$

The definition of \mathcal{E} by case distinction on the last rule in a proof π can be found in Table 1. The intuition behind the definition is the following: a proof of a conjunction contains computational content for both its subproofs, a proof of a disjunction contains information which disjunct is true, and the computational content of that disjunct, a proof that eliminates a disjunction corresponds to a case distinction on whether the left or right conjunct is true (and passes on the computational content of the proof of this disjunct), and so on.

Instead of going the usual way of proving correctness of the translation, we will instead apply these definitions to an example and verify that indeed, witness-computing programs are extracted.

Example. We show how to synthesize the maximum function $\max : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ from its specification in our setting. Letting

$$F(x_1, x_2, y) = y \geq x_1 \wedge y \geq x_2 \wedge (y = x_1 \vee y = x_2),$$

the max function has the specification $\forall x_1, x_2 \in \mathbb{N} : F(x_1, x_2, \max(x_1, x_2))$. We extract a functional program realizing max from a natural deduction proof of $\forall x_1, x_2 \exists y.F(x_1, x_2, y)$ using the lemma $L := \forall x_1, x_2.x_1 \geq x_2 \vee x_2 \geq x_1$. Let π be the proof

$$\frac{\frac{(\psi) \quad \forall x_1, x_2.x_1 \geq x_2 \vee x_2 \geq x_1}{x_1 \geq x_2 \vee x_2 \geq x_1} \forall_e \quad \frac{\frac{x_1 \geq x_1 \quad [x_1 \geq x_2] \quad \frac{x_1 = x_1}{x_1 = x_1 \vee x_1 = x_2} \vee_i}{F(x_1, x_2, x_1)} \exists_i \quad \frac{[x_2 \geq x_1] \quad x_2 \geq x_2 \quad \frac{x_2 = x_2}{x_1 = x_2 \vee x_2 = x_2} \vee_i}{F(x_1, x_2, x_2)} \exists_i}{\frac{\exists y.F(x_1, x_2, y)}{\forall x_1, x_2 \exists y.F(x_1, x_2, y)} \forall_i} \vee_e$$

³Note that this is an important distinction between the interpretation of proofs as functions in our setting, and the setting of the Curry-Howard isomorphism as it is classically understood: in that setting, the proofs of $A \rightarrow A$ are exactly the programs $\mathbb{N} \rightarrow \mathbb{N}$.

⁴In the typed setting, one would introduce an accompanying type ε for “no computational content”, and one would propagate this information as much as possible. E.g. one would identify the types $A \rightarrow \varepsilon$ and ε , since $A \rightarrow \varepsilon$ would be the type of a function taking an object of type A , and returning something which does not have computational content. Functions of such types may appear when functional extraction is done naively, and one wants to avoid creating λ -terms of such types for efficiency reasons.

⁵As expected, the computational content intuitively corresponds to the witness of the $\exists y$ quantifier (structured in the correct way — compare this with the definitions in Table 1). But note that \div is not part of the language used in our proof system: it is only contained in our programming language. Still, we have $\mathbb{N} \models t = 0 \vee t = (t \div 1) + 1$ as expected. This is an example of the general observation that all axioms which have a computational interpretation in the programming language can be added to the proof system.

π	$\mathcal{E}(\pi)$
$\frac{(\pi_1) \quad (\pi_2)}{\frac{A \quad B}{A \wedge B} \wedge_i}$	$(\mathcal{E}(\pi_1), \mathcal{E}(\pi_2))$
$\frac{(\pi_1)}{\frac{A}{A \vee B} \vee_i}$	$(0, \mathcal{E}(\pi_1))$
$\frac{(\pi_1)}{\frac{B}{A \vee B} \vee_i}$	$(1, \mathcal{E}(\pi_1))$
$\frac{[A] \quad [B]}{\frac{(\pi_1) \quad \vdots(\pi_2) \quad \vdots(\pi_3)}{\frac{A \vee B \quad C \quad C}{C} \vee_e} \vee_e$	If $\text{left}(\mathcal{E}(\pi_1))$ Then $\mathcal{E}(\pi_2)[x_A := \text{right}(\mathcal{E}(\pi_1))]$ Else $\mathcal{E}(\pi_3)[x_B := \text{right}(\mathcal{E}(\pi_1))]$
$\frac{(\pi_1) \quad (\pi_2)}{\frac{A \quad A \rightarrow B}{B} \rightarrow_e}$	$\mathcal{E}(\pi_2)\mathcal{E}(\pi_1)$
$\frac{(\pi_1)}{\frac{A}{\forall x.A} \forall_i}$	$\lambda x. \mathcal{E}(\pi_1)$
$\frac{(\pi_1)}{\frac{\forall x.A}{A[x := t]} \forall_e}$	$\mathcal{E}(\pi_1)t$
$\frac{[B]}{\frac{(\pi_1) \quad \vdots(\pi_2)}{\frac{\exists x.B \quad A}{A} \exists_e} \exists_e$	$\mathcal{E}(\pi_2)[x := \text{left}(\mathcal{E}(\pi_1))][x_B := \text{right}(\mathcal{E}(\pi_1))]$
$\frac{(\pi_1)}{\frac{A[x := t]}{\exists x.A} \exists_i}$	$(t, \mathcal{E}(\pi_1))$
$\frac{[A(n)]}{\frac{(\pi_1) \quad \vdots(\pi_2)}{\frac{A(0) \quad A(n+1)}{\forall n.A} \text{IND}} \text{IND}$	$\lambda u. \mathbf{R}(\mathcal{E}(\pi_1))(\lambda n \lambda x_{A(n)}. \mathcal{E}(\pi_2))u$
$\frac{(\pi_1) \quad (\pi_2)}{\frac{t = s \quad A(s)}{A(t)} =}$	$\mathcal{E}(\pi_2)$

Table 1: Extraction of computational content from proofs.

For the moment, we omit how exactly the proof (ψ) of the lemma L is treated. We construct the computational content of π according to the interpretation of the leafs and Table 1. Letting f denote the computational content of ψ , and letting (for easier readability) $x_{x_1 \geq x_2} = Y$ and $x_{x_2 \geq x_1} = Z$, we obtain the following. The left \exists_i induces the λ -term $(x_1, (\varepsilon, (Y, (0, \varepsilon))))$ and the right \exists_i induces the λ -term $(x_2, (Z, (\varepsilon, (1, \varepsilon))))$. Putting things together, we obtain for π the λ -term $\mathcal{E}(\pi)$:

$$\lambda x_1 x_2. \mathbf{If} \mathbf{left}(f x_1 x_2) \mathbf{Then} (x_1, (\varepsilon, (\mathbf{right}(f x_1 x_2), (0, \varepsilon)))) \mathbf{Else} (x_2, (\mathbf{right}(f x_1 x_2), (\varepsilon, (1, \varepsilon)))).$$

Now let α_1, α_2 be numerals. Assuming that f is interpreted correctly (i.e. that $\mathbf{left}(f \alpha_1 \alpha_2)$ normalizes to 0 if $\alpha_1 \geq \alpha_2$ and 1 otherwise), the term for π , when applied to α_1, α_2 , normalizes correctly either to (α_1, \dots) or (α_2, \dots) , where \dots contains the computational content of the conjuncts of $F(\alpha_1, \alpha_2, \alpha_i)$ (which in this case is anyways empty since $F(x, y, z)$ is quantifier-free). Hence $\mathbf{left}(\mathcal{E}(\pi) \alpha_1 \alpha_2)$ computes $\max(\alpha_1, \alpha_2)$ as desired.

Regarding the proof ψ of L , we have two options: either we assume that we have a program that, given $\alpha_1, \alpha_2 \in \mathbb{N}$ decides whether $\alpha_1 \geq \alpha_2 \vee \alpha_2 \geq \alpha_1$ (in this case, we treat L as an axiom), or we prove L and synthesize the program from the proof. In practice, the first option is more reasonable, but for sake of exposition we take the second option here: indeed, the proof of the lemma involves induction and therefore gives rise to a recursive program. Setting $A(x_1) := \forall x_2. x_1 \geq x_2 \vee x_2 \geq x_1$, we let ψ be

$$\frac{(\psi_b) \quad (\psi_s)}{\forall x_1 A(x_1)} \text{IND}$$

where ψ_b is

$$\frac{\frac{x_2 \geq 0}{0 \geq x_2 \vee x_2 \geq 0} \vee_i}{A(0)} \forall_i$$

and ψ_s is

$$\frac{\frac{x_2 = 0 \vee \exists y. x_2 = y + 1 \quad (\varphi_l) \quad (\varphi_r)}{n + 1 \geq x_2 \vee x_2 \geq n + 1} \vee_e}{A(n + 1)} \forall_i$$

where φ_l is

$$\frac{[x_2 = 0] \quad \frac{n + 1 \geq 0}{n + 1 \geq 0 \vee x_2 \geq n + 1} \vee_i}{n + 1 \geq x_2 \vee x_2 \geq n + 1} =$$

and φ_r is

$$\frac{[\exists y. x_2 = y + 1] \quad \frac{[x_2 = y + 1] \quad (\varphi)}{n + 1 \geq x_2 \vee x_2 \geq n + 1} =}{n + 1 \geq x_2 \vee x_2 \geq n + 1} \exists_e$$

where φ is

$$\frac{\frac{[A(n)]}{n \geq y \vee y \geq n} \forall_e \quad \frac{\frac{[n \geq y] \quad n \geq y \rightarrow n + 1 \geq y + 1}{n + 1 \geq y + 1} \rightarrow_e \quad \frac{[y \geq n] \quad y \geq n \rightarrow y + 1 \geq n + 1}{y + 1 \geq n + 1} \rightarrow_e}{\frac{n + 1 \geq y + 1 \vee y + 1 \geq n + 1}{n + 1 \geq y + 1 \vee y + 1 \geq n + 1} \vee_i \quad \frac{y + 1 \geq n + 1}{n + 1 \geq y + 1 \vee y + 1 \geq n + 1} \vee_i}{n + 1 \geq y + 1 \vee y + 1 \geq n + 1} \vee_e$$

Setting $x_{A(n)} = Z$ and $x_{x_2 = y + 1} = U$ for readability, functional extraction yields $\mathcal{E}(\psi)$ (after application of some reduction rules to improve readability):

$$\underbrace{\lambda u. \mathbf{R}(\underbrace{\lambda x_2. (1, \varepsilon)}_{\mathcal{E}(\psi_b)}) (\underbrace{\lambda n \lambda Z \lambda x_2. \mathbf{If} \mathbf{isZero}(x_2) \mathbf{Then} \underbrace{(0, \varepsilon)}_{\mathcal{E}(\varphi_l)} \mathbf{Else} \underbrace{\mathbf{If} \mathbf{left}(Z(x_2 \div 1)) \mathbf{Then} (0, \varepsilon) \mathbf{Else} (1, \varepsilon)}_{\mathcal{E}(\varphi_r) = \mathcal{E}(\varphi)[y := x_2 \div 1][U := \varepsilon]})}_{\mathcal{E}(\varphi_s)}}_{\mathcal{E}(\psi)} u$$

One can check that for all $\alpha_1, \alpha_2 \in \mathbb{N}$, we have that if $\mathcal{E}(\psi)_{\alpha_1 \alpha_2}$ reduces to $(0, \dots)$, then $\alpha_1 \geq \alpha_2$, and if it reduces to $(1, \dots)$, then $\alpha_2 \geq \alpha_1$, and that this term always reduces to one of these two forms. Note that the length of the reduction sequence is linear in α_1 since we recurse from $\alpha_1 + 1$ to α_1 . A logarithmic algorithm (corresponding to the comparison of the binary representations of α_1, α_2) could be obtained by using “binary induction” $A(0) \wedge (\forall x. A(x) \rightarrow A(0x) \wedge A(1x)) \rightarrow \forall x A(x)$.

Soundness. For our purposes, the most important notion of soundness is that from proofs of Π_2 -statements, i.e. statements of the form $\forall x \exists y. F(x, y)$, with $F(x, y)$ quantifier-free, we can extract programs that compute a correct y given an x , as indicated above. To do this, one would define a binary relation “ t realizes F ”, where t is a λ -term and F is a formula, by structural induction on F . In particular, the definition would ensure that if F is a Π_2 -statement and t realizes F , then t is a suitable program. One would finally show, by induction on natural deduction proofs π , that indeed $\mathcal{E}(\pi)$ realizes F , where F is the formula that π proves. We refer to the literature for more details.

Outlook. There are many directions one can go from here. Note that we have only treated intuitionistic arithmetic — classical arithmetic can be treated by embedding it into intuitionistic logic (using e.g. a double-negation translation etc.), or directly by interpreting classical proofs or the law of excluded middle. We have not even given a computational interpretation for all the usual rules of intuitionistic natural deduction (only what we used in our example proof) — one could interpret the whole system. Alternatively, one could just show how to interpret minimal logic (where the only connective is \rightarrow), and embed intuitionistic into minimal logic. One can investigate how the translation can be improved by removing redundant parts of the extracted program (this can prevent construction of a term that is never used computationally in the proof).